MUON $g - 2$

OFFLINE COMPUTING AND SOFTWARE MANUAL [GM2 V6_01_00]

May 5, 2015 `Git version: v6_01_00_00-0-g30fa9a9` GM2-doc-1825

Contents

1

What is this document?

This document is meant to be a user's manual to the Muon g-2 offline and simulation software and computing system.[1] This document is a PDF file, so it is trivial to search and you can copy it to your computer/tablet/phone/watch and read it anywhere including your office, in meetings, on the plane, in the tub, etc. It is also generated by a git repository using the same build system infrastructure as our code base, so it is easy to version itself and keep in sync with code versions. We support writing sections directly in LaTeX (which you probably already know) and in Markdown (like LaTeX, but simpler). Finally, there's a special script that can run shell commands and put the output directly in the document (no cutting and pasting).

[1] This document replaces the documentation we had in the Redmine Wiki because the Wiki was hard to edit and keep up-to-date, hard to sync with versions, hard to search, and required a network connection.

The idea is to have documentation that is easy to read, easy to write, and easy to keep up to date. All links in the document are click-able in your PDF reader.

One nice thing about having Wiki pages was that each page can be short and so the documentation looks manageable, until you try to find something. The problem with one big PDF file is that it will be big and will look overwhelming. Remember to read the section titles carefully and just read what you need. Furthermore, all of the links to sections (e.g. in the table of contents) are live and will allow you to navigate the file easily. Nearly every PDF reader has a back button to take you back to previously read pages (back-traversing links if necessary); it will probably come in handy.

1.1 What code goes with this document?

The title of this document states the corresponding version of `gm2`. `gm2` is the "umbrella" product that specifies a release. For example, this version of the document goes with `gm2 v6_01_00`. On the bottom of the title page is the git version information for this document itself. For this version, it reads `v6_01_00_00-0-g30fa9a9`. There are three or four parts to this description, separated by *dashes* (not underscores; the underscores are part of the version). The first part corresponds to the `gm2` version, with an additional two digits at the end since the documentation may be updated more often than the g-2 code. This version should be the git tag of this document. The second part is the number of commits past the tag. If it is non-zero, then there are untagged changes. The third part is `g` followed by the git hash of the commit corresponding to this document (e.g. `0be91c0`). All of this could be followed by `-dirty`, which means that this document comes from source files with uncommitted changes.[2]

[2] Official documentation has zero for the second part (number of untagged commits) and no `-dirty`.

1.2 *Obtaining this documentation*

The latest official version of this documentation is in GM2 DocDB as GM2-DOC-1825.[3] Newer releases of `gm2` (staring from `gm2 v5_01_00`) will have a copy of this manual that corresponds to the particular `gm2` version at `$GM2SWDOCS_DIR/manual.pdf`.

[3] DocDB uses its own versioning scheme (just a sequential number) which does not correspond to the `gm2` release.

1.3 *Obtaining the source for this documentation, contributing to it, and building it*

To get the source,[4] follow the instructions in section 3. When you get to section 3.3, instead of checking out `gm2artexamples`, checkout `gm2swdocs`. You will be in the `develop` branch. If you want to checkout a particular tag, branch, or hash, you can do that with the `git checkout` command. For example,

[4] *Note:* The program `pandoc` at http://johnmacfarlane.net/pandoc is used to convert markdown and other file formats to LaTeX. It is part of our g-2 release for SLF6 machines. See below for installing it on your own machine.

```
git tag                         # Show all of the tags
git checkout v5_00_00_02   # Check out sources for this tag
```

You can also do `git checkout` on a git commit hash value to checkout the sources for that particular commit.

1.3.1 *Changing and adding to documentation*

If you want to change or add documentation, you should start a feature branch with `git flow feature start <your_branch_name>`. You can then alter or add your own documentation. When you are ready to complete your feature branch, send mail to `gm2-sim@fnal.gov` and let people look at your changes first.

There are several directories in `gm2swdocs`. You should not need to alter anything in the `Modules` nor `ups` directories. The former contains cmake macros needed for building the source files into PDF. The latter is for the build and release system. The other directories, `latex`, `markdown`, `bashmd` is where you'll put your documentation or make changes.

The latex directory has files in LaTeX as well as some LaTeX infrastructure files. The most important file in there is manual.tex, which is the main driver file for this document.[5] All other parts come in with an \include{filename.tex} command, but this is handled automatically by a cmake variable (you won't see the \include lines in the file). If you add your own LaTeX file in the latex directory, follow instructions in srcs/gm2swdocs/CMakeLists.txt.

[5] We are using a document class based on "Tufte" documents, where notes and captions go into the wide right margin. Please see the existing LaTeX files for examples.

The markdown directory has files written in the Markdown format and converted by Pandoc. A Google search on Markdown will give you lots of information. The Pandoc variant of Markdown is described in http://johnmacfarlane.net/pandoc/demo/example9/pandocs-markdown.html. See existing files in this directory for examples. If you want to write something quickly and do not need fancy LaTeX, then Markdown is the way to go. If you add a file to this directory, you must follow the instructions in markdown\CMakeLists.txt.

The bashmd directory has files written in Markdown but also actually runs bash code with the output going into the document. The best file to look at for an example is bashmd/gettingStarted_gm2artexamples.bashmd. Again, if you add a file to this directory, see bashmd/CMakeLists.txt for instructions.

Pandoc understands many Wiki mark-up formats. If you have a favorite one, it is possible to add it to this document and have pandoc process it. Ask for help. If you are not passionate about mark-up formats, then please just use Markdown as it works very well.

1.3.2 Building the documentation

If you are on Mac, a Windows machine, or your own Linux machine, you must have installed a full TeX suite and pandoc on your system. See http://johnmacfarlane.net/pandoc/installing.html for installation instructions for pandoc. If you are on gm2gpvm, everything is installed there for you, but you must issue setup pandoc; see below.

Assuming your environment is set up (see above) then you need to do, once per session, . mrb s . If you are on gm2gpvm, do setup pandoc (it only works on SLF6, so use machines gm2gpvm02-04). Then you can do mrb b to build. Note that by default, files in bashmd/ will *not* be built as they can take a long time. If you do want them built, then do mrb b -DBUILD_BASHMD=1. Also, pdflatex will run many times to ensure that references and table of contents are all resolved. If you make changes, only those changed files will be rebuilt on subsequent builds. If you see an error like Cannot find PANDOC and you are own gm2gpvm, then you forgot to issue the setup pandoc command.

The output PDF file will be in `$MRB_BUILDDIR/gm2swdocs/latex/manual.pdf`. On a Mac, you can view it with,

```
open $MRB_BUILDDIR/gm2swdocs/latex/manual.pdf
```

When you have completed your feature branch, send mail to `gm2-sim@fnal.gov` and await further instructions.

2
Releases of gm2

This section describes the various releases of gm2. At the end of this section is a description of the release philosophy. Some releases will have a `debug` build. You should only use those builds for debugging. Use the `prof` build for analyses.

The constituent versions of main packages are given for each `gm2` release. You can see a list of all dependencies by running `ups depend`. For example,

```
ups depend gm2 v6_01_00 -q prof
```

For $g-2$ products, you can usually see a change log in the product directory. For example,

```
less $GM2RINGSIM_DIR/CHANGELOG
```

Special migration instructions are given where necessary. Migration in general is covered in the developer workflow section.

2.1 `gm2 v6_01_00 -q prof` and (`-q debug`)

This is the first point release of the `v6` series.

Contains:

- `gm2ringsim v3_00_00`
- `gm2geom v3_00_00`
- `gm2dataproducts v3_00_00`
- `artg4 v3_00_00`
- `gm2artexamples v3_00_00`

2.2 `gm2 v6_00_00 -q prof` and (`-q debug`)

This a base release of the `v6` series. Note that you no longer have to specify the qualifier for the compiler version (is `e7` for this release)[1]

[1] There is only one compiler version that works for this release.

As per the release philosophy, there are no g-2 packages in this release, only externals are released.

`gm2 v6_00_00` has the following:

- `art v1_13_01` Release Notes
- `root v5_34_25` Release Notes
- `geant v4_9_6_p04a` 4.9.6, p01, p02, p03, p04
- `gcc v4_9_2` with `-std=c++1y` for C++14 features.
- `gsl v1_16` (GNU scientific library) (new!)

This release works on the following platforms:

- Scientific Linux 5
- Scientific Linux 6
- Mac OSX Mavericks
- Mac OSX Yosemite (new!)

Note that Mac OSX Mountain Lion is no longer supported.

There are significant improvements that speed up builds, including a replacement for `make` called `ninja`. See section 4.1.

2.2.1 How to migrate from `v5_XX_XX` to `v6_00_00`

Updating source code: The `develop` branches of the standard simulation packages are now compatible with `v6_00_00`. If you have a branch that you need to update, you can merge `develop` on to your branch with,

```
git merge develop
```

Be sure to read the next section for important changes.

Building with `v6_00_00`: In general, it is easiest to start with a new development area. You can re-use an old one by setting up the $g-2$ environment, explicitly setting up this version of `gm2` and then, in the top level directory of your development area, do

```
mrb newDev -p
```

That command will make a new `localProducts...` area. You must source the setup script in there to continue.

Missing symbols involving `TFileService`: The `TFileService` has changed and now involves a template, which means one must explicitly link in its library.

If you have a module that uses a `TFileService`, find the `CMakeLists.txt` file and add to the `art_make` after

`MODULE_LIBRARIES`, `art_Framework_Services_Optional_TFileService_service`.
For example,

```
# Fill Builder Filter Module
art_make( MODULE_LIBRARIES
          gm2analyses_calo_clustering
          art_Framework_Services_Optional_TFileService_service
        )

# Install header files into the products area
install_headers()

# Don't do clustering until we get Nic's CaloGeometry_service up and running again
add_subdirectory( clustering )
```

I have made this change for most of our packages in the `feature/gm2v6` branch (now in `develop`).

Problems with Root on Mac Yosemite: The version of root setup by `gm2 v6_00_00` (or any v6 series release) gives an error when you try to open a `TBrowser` (this only happens on the Mac Yosemite platform). An updated root for yosemite is available. When you are ready to analyze data with root, do the following beforehand:

```
setup root v5_34_25a -q e7:prof
```

This will give you a version of root that works on Yosemite.

2.3 gm2 v5_01_00 -q e6:prof

This is the first point release of the `v5` series.
Contains:

- `gm2ringsim v2_00_00`
- `gm2geom v2_00_00`
- `gm2dataproducts v2_00_00`
- `artg4 v2_00_00`
- `gm2artexamples v2_00_00`

2.4 gm2 v5_00_00 -q e6:prof and (-q e6:debug)

Note the new version numbering scheme as per the release philosophy. This release is the fifth one for g-2 since time began, thus the `v5`.
This release is the base release of the v5 series.
`gm2 v5_00_00` has the following:

- `art v1_12_02` Release Notes
- `root v5_34_21b` Release Notes
- `geant4 v4_9_6_p03e` Release Notes: 4.9.6, p01, p02, p03
- `gcc v4_9_1` with `-std=c++1y` for C++14 features.

2.4.1 *How to migrate from `v201402` to `v5_00_00`*

If you have a branch that works with `gm2 v201402`, then you will need to make some changes for it to work in `gm2 v5_00_00` as some parts of the build system have changed. If you are working on code we have in Redmine, and you can merge the `develop` branch onto your branch without breaking your code. Do the following:

```
# Go to your source directory and check out your branch
# Check in all code you've been working on and push to Redmine

# Now merge develop onto your branch
$ git pull origin develop
```

If there are merge conflicts, then you will have to resolve them. Accept changes from `develop` for `CMakeLists.txt` files and `product_deps` as those will have the necessary changes.

2.5 *`gm2 v201402 -q e4:prof`*

`gm2 v201402` has the following:

- `art v1_08_10` Release Notes
- `root v5_34_12`
- `geant4 v4_9_6_p02`
- `gcc v4_8_1` with `-std=c++11` for C++11 features.
- `cmake v2_8_8`

and

- `gm2ringsim v1_00_00`
- `gm2geom v1_00_00`
- `gm2dataproducts v1_00_00`
- `artg4 v1_00_00`

This is the old release with the old date scheme. It should no longer be used.

2.6 *The Release Philosophy*

What is a `gm2` release? A `gm2` release is a versioned collection of libraries and executables that you either use or build your code against.

A particular `gm2` release contains a particular version of `gcc`, `art`, `root`, `geant4`, etc. These libraries/executables are called the *externals*. A `gm2` release may also contain $g-2$ applications and libraries built against those externals (e.g. `gm2ringsim`, `artg4`). If the versions of those packages are suitable for you, then you can use them directly without having to build them yourself. This means we have *official* versions of these packages.

Official releases are important. For the purpose of scientific reproducibility, it is important to know how results were produced. Using a versioned release means that we know the code used for an analysis and can re-run it to do further analyses or look for mistakes. Official releases are essential for sharing code, as is gives people a common base and starting place.

The philsophy of `gm2` releases is that the first (major) version number in the release (e.g. the 5 in `v5_00_00`) is the release *series*. Releases in the same series are built with the same version of externals (`gcc`, `art`, `geant4`, `root`, etc) and so they are all binary compatible. The `vX_00_00` release only has externals in it and is called a *base* release. We then add point releases (e.g. `v5_01_00`, `v5_02_03`) containing $g-2$ libraries and applications (e.g. `gm2ringsim`). If there is a new `art` or `root`, then the major version number increases (e.g. to `v6_00_00`) and a new series is started. New major releases (new series) should occur only 3-4 times per year.

The point releases can occur more often and represent official changes to the $g-2$ code base (e.g. new geometry or features in the simulation). Feature changes advance the middle (minor) version number and bug fixes advance the last (patch) version number. So the first release of $g-2$ code for a new series is `vX_01_00`. A feature addition will advance to `vX_02_00`. A subsequent bug fix will advance to `vX_02_01`.

Users adoption of these point releases is optional. They can always build all of the necessary code based on the `vX_00_00` base release. But using a point release can be convenient and save a large amount of time by using pre-built libraries instead of building them by hand. The point releases also represent a trackable official progression of features and bug fixes. Users can also use libraries from a point release, but build parts of the release themselves that they are developing (e.g. developing `gm2ringsim`, which is also in the release). In these cases, the build system will automatically use the user developed code instead of what is in the release. The `superbuild` system, which does builds across platforms for use on the grid, will automatically mark such user-built libraries as unofficial.

2.6.1 How do releases help me as a user/developer?

If the official released code is suitable for you (e.g. you are not developing the code in the release, but are instead developing code that *uses* the release), then using an official release will save you compilation time and will be more convenient. You can more easily track what you have run on the grid. You may be able to run without having to build anything (e.g. simply running the official `gm2ringsim`.

You should use an official point release whenever possible. Instructions for setting up your development area and how to migrate to new releases are given in the section under the release notes as well as in the developer workflow section of this manual.

3
Getting started with gm2artexamples

This section is a short tutorial to show you quickly how to get started by,

- Logging in and selecting a release (the latest)
- Starting a development area
- Checking out code (`gm2artexamples`)
- Building it
- Testing
- Running
- Logging in again

For this tutorial, we'll use the `gm2artexamples` product.[1] This is a good product to use if you are getting started.

[1] We use the terms *product*, *project*, and *package* somewhat interchangeably. All of our products live on the Redmine server, http://redmine.fnal.gov

3.1 Logging in and selecting a release area

Fermilab has several interactive virtual machines for use by the Muon $g-2$ collaboration. See here for more information about how to log in. Our releases (libraries, executables) are served by CVMFS.[2] CVMFS is already mounted on the Fermilab interactive VMs. If you have a Mac, you can install CVMFS yourself by looking here, and then use your Mac to develop code.

[2] CVMFS is a system that serves application code and updates automatically when new files are released.

Once you've logged into the machine, you need to select a release area. You *always*[3] need to do this step everytime you log in. If you are on a Fermilab interactive VM (`gm2gpvm01`, `gm2gpvm02`, `gm2gpvm03`, `gm2gpvm04`), you select the release area by doing,

[3] You need to do this step everytime you log in because you can use different release areas for the same development area, say, for example, if CVMFS is down or you are sharing a directory between your Mac and a Linux system.

```
$ source /grid/fermiapp/gm2/setup # On gm2gpvm machine
```

Note that `$` is the shell prompt (don't type it in).

If you are on a Mac or another system with CVMFS OASIS installed, you do,

```
$ source /cvmfs/oasis.opensciencegrid.org/gm2/prod/g-2/setup
```

Now we'll actually run it so you can see the output. This script will work on both Mac and gm2gpvm. You may want to put it in your .profile on gm2gpvm.

```
$ if [ -r /grid/fermiapp/gm2/setup ]; then  # Does /grid/fermiapp/gm2/setup exist?
$   source  /grid/fermiapp/gm2/setup   # We're on gm2gpvm
$ else
$   source /cvmfs/oasis.opensciencegrid.org/gm2/prod/g-2/setup # We're on a Mac
$ fi
```

```
g-2 software

--> To list gm2 releases, type
ups list -aK+ gm2

--> To use the latest release, do
setup gm2 v6_00_00 -q prof

For more information, see https://cdcvs.fnal.gov/redmine/projects/g-2/wiki/ReleaseInformation
```

You may want to put that source command in your ~/.profile file. Furthermore, if you are on a gm2gpvm machine, you should also put setup git at the bottom of your ~/.profile.

3.2 Starting a development area

Now that the release area is selected, you need to make a *development area.* The development area contains source code, build products, and a personal release area. You typically use a development area for a particular topic, such as adding a feature to the simulation or generating a plot for some study. You can have as many development areas as you want, but only one can be active at a time.

Make an empty directory and go there. If you are on a gm2gpvm machine, you should make an area in /gm2/app/users/<YOUR_NAME>.[4] You can put code in your home directory, but that has a small quota and you can easily use it all up. There is no quota on /gm2/app, but it is not backed up.

[4] If this directory does not exist, you can make it with the mkdir command.

```
$ mkdir /gm2/app/users/lyon/first-try  # On gm2gpvm
$ cd /gm2/app/users/lyon/first-try
```

If you are on your Mac, or some other machine, make the directory where you have room. Here's a script that will run on both Mac and gm2gpvm.

```
$ if [ -r /gm2/app/users/$USER ]; then  # Does /gm2/app/users/YOU exist?
$   # It does, let's use /gm2/app/users/$USER/first-try followed by random letters for uniqueness
$   TMPDIR=`mktemp -d /gm2/app/users/$USER/first-try.XXXX`
$ else
$     # We're not on gm2gpvm, let's just make a directory in your home area (hope there's room!)
$     TMPDIR=`mktemp -d ~/first-try.XXXX`
$ fi
$
$ # Change directory there
$ cd $TMPDIR
```

Note that all subsequent commands are the same for `gm2gpvm` and Mac or whatever.

Since you are starting out with a new area, you must choose a release. You should generally choose the latest, which will be specified in the output when you selected the release area. Just do what the command says,[5]

[5] `setup` is a *ups* command. UPS is our release and product management system.

```
$ setup gm2 v6_01_00 -q prof
```

So here we are setting up g-2 release `v6_01_00` with the `prof` qualifier. `prof` means we'll do a profile build. Profile builds are optimized and have debugging symbols turned on.

Now, you must create the development area. You will start using the `mrb` commands. *mrb* means "multi-repository build system" and is a build system used by Muon $g-2$, the art developers, and LBNF. You can get a list of `mrb` commands with (you don't have to type in the full path that you see below),

```
$ mrb -h
Usage /cvmfs/oasis.opensciencegrid.org/gm2/prod/external/mrb/v1_03_00a_gm2/bin/mrb  [-h for help]"

  Tools ( for help on tool, do "/cvmfs/oasis.opensciencegrid.org/gm2/prod/external/mrb/v1_03_00a_gm2/bin/mrb <tool> -h" )

    newDev (n)                  Start a new development area
    gitCheckout (g)             Clone a git repository
    svnCheckout (svn)           Checkout from a svn repository
    setEnv (s)                  Setup development environment (mrbSetEnv)
    build (b)                   Run buildtool
    install (i)                 Run buildtool with install
    test (t)                    Run buildtool with tests
  superbuild (sb)             Run superbuild on the Jenkins buildmaster
    setup_local_products (slp)  Setup local products (mrbslp) [not local sources]
    zapBuild (z)                Delete everything in your build area
    newProduct (p)              Create a new product from scratch
    changelog (c)               Display a changelog for a package
    bumpVersion (bv)            Bump version number of a package
    updateDeps (ud)             Update dependencies in CMakeLists.txt and product_deps
    updateCM (uc)               Update the master CMakeLists.txt file
  updateLocalProdDeps (ulpd)  Update dependencies in product_deps to reflect local products
    makeDeps (md)               Build or update a header level dependency list
    checkDeps (cd)              Check for missing build packages
    pullDeps (pd)               Pull missing build packages into MRB_SOURCE

  Aliases ( we use aliases for these commands because they must be sourced )

   mrbsetenv                   Setup a development enviornment and local products  [use this more often]
                               (source $MRB_DIR/bin/mrbSetEnv)

   mrbslp                      Setup only the products installed in the working localProducts_XXX directory
                               (source $MRB_DIR/bin/setup_local_products)
```

The `mrb` commands are the same if you are on `gm2gpvm` or your Mac.

To initialize your development area, do this in an empty directory.

```
$ mrb newDev
```

```
building development area for gm2 v6_01_00 -q prof

MRB_BUILDDIR is /Users/lyon/first-try.Wrih/build_d14.x86_64
MRB_SOURCE is /Users/lyon/first-try.Wrih/srcs
INFO: cannot find releaseDB/base_dependency_database
      mrb checkDeps and pullDeps may not have complete information
MRB_PROEJCT IS gm2

IMPORTANT: You must type
    source "/Users/lyon/first-try.Wrih/localProducts_gm2_v6_01_00_prof/setup"
NOW and whenever you log in
```

Read the output carefully. Some things to note:

- A build directory is created and note its name contains the flavor of your machine.[6] You can get to that directory easily with `cd $MRB_BUILDDIR` .
- A source directory is created for your source code. You can get to it easily by doing `cd $MRB_SOURCE` .
- You can ignore the message about the release database. That's a LBNF thing we don't use.
- The important message is indeed important. There is a set up script that you need to run that sets up your environment. Run that script now and whenever you log in to restore your development environment. You don't need to type in the whole path, since you are at the top of your development area.

```
$ source localProducts_gm2_v6_01_00_prof/setup
```

```
MRB_PROJECT=gm2
MRB_PROJECT_VERSION=v6_01_00
MRB_QUALS=prof
MRB_TOP=/Users/lyon/first-try.Wrih
MRB_SOURCE=/Users/lyon/first-try.Wrih/srcs
MRB_BUILDDIR=/Users/lyon/first-try.Wrih/build_d14.x86_64
MRB_INSTALL=/Users/lyon/first-try.Wrih/localProducts_gm2_v6_01_00_prof

PRODUCTS=/Users/lyon/first-try.Wrih/localProducts_gm2_v6_01_00_prof:/cvmfs/oasis.opensciencegrid.org/gm2/prod/g-2:/Users/lyon/Development/g-2/docs/localProducts_gm2_v6_01_00_prof:/cvmfs/oasis.opensciencegr
```

- A local products area is also created. This is your own personal release area that overlays the official one (so stuff you have in your personal release area override products in the official one).

3.3 Checkout code

Now you need to checkout some code. For this example, we'll use the `gm2artexamples` product. All of our code lives in `git` repositories on http://redmine.fnal.gov . The `mrb gitcheckout` command is used to clone the git repositories (this is a convenience command so you don't have to remember the git URLs and other set up tasks).[7] Let's check out the `gm2artexamples` product. You must be in the `srcs` directory of your development area. The command is rather chatty.

[6] Mac is d13 (for Darwin version 13) and slf5, slf6 are marked as appropriate.

[7] You can type `mrb g` for short.

```
$ cd srcs
$ mrb g gm2artexamples
$ cd gm2artexamples
$ cd ..
```

```
git clone: clone gm2artexamples at /Users/lyon/first-try.Wrih/srcs
NOTICE: Running git clone ssh://p-gm2artexamples@cdcvs.fnal.gov/cvs/projects/gm2artexamples
Cloning into 'gm2artexamples'...
X11 forwarding request failed on channel 0
ready to run git flow init for gm2artexamples
Already on 'master'
Your branch is up-to-date with 'origin/master'.
Using default branch names.
Already on 'develop'
Your branch is up-to-date with 'origin/develop'.
Branch develop set up to track remote branch develop from origin.
X11 forwarding request failed on channel 0
Already up-to-date.
NOTICE: Adding gm2artexamples to CMakeLists.txt file
NOTICE: You can now 'cd gm2artexamples'

You are now on the develop branch (check with 'git branch')
To make a new feature, do 'git flow feature start <featureName>'
```

If you have more code to checkout, then run more `mrb g` commands.

3.4 Building code

Now that your code is checked out, you need to build it. The first step you need to do is to "extend" your environment with any products your build depends upon set up. The way to do this is to do `source mrb setEnv`.[8] You need `source` (or `.` for short) because your shell environment needs to be extended with new environment variables. You need to run this command after you log back into and start developing. If you do not make major changes to your code (you don't introduce new dependencies), then you only need to run the command once before you build.

[8] There are two shortcuts for `source mrb setenv`; you can do `. mrb s` or `mrbsetenv` (the latter is a bash function that does the `source` for you).

```
$ . mrb s
```

```
local product directory is /Users/lyon/first-try.Wrih/localProducts_gm2_v6_01_00_prof
----------- this block should be empty ------------------
---------------------------------------------------------
The working build directory is /Users/lyon/first-try.Wrih/build_d14.x86_64
The source code directory is /Users/lyon/first-try.Wrih/srcs
----------- check this block for errors -----------------------
----------------------------------------------------------------
```

You should not see any errors between the dashed lines. If you do, then you have some product dependency mismatch (ask for help).

Now you can build your code. The build command is `mrb build`.[9]

[9] `mrb b` for short

```
$ mrb b
```

```
/Users/lyon/first-try.Wrih/build_d14.x86_64
calling buildtool -I /Users/lyon/first-try.Wrih/localProducts_gm2_v6_01_00_prof -b
INFO: Install prefix = /Users/lyon/first-try.Wrih/localProducts_gm2_v6_01_00_prof
INFO: CETPKG_TYPE = Prof
```

```
------------------------------------
INFO: Stage cmake.
------------------------------------

-- The C compiler identification is GNU 4.9.2
-- The CXX compiler identification is GNU 4.9.2
-- Checking whether C compiler has -isysroot
-- Checking whether C compiler has -isysroot - yes
-- Checking whether C compiler supports OSX deployment target flag
-- Checking whether C compiler supports OSX deployment target flag - yes
-- Check for working C compiler: /cvmfs/oasis.opensciencegrid.org/gm2/prod/external/gcc/v4_9_2/Darwin64bit+14/bin/gcc
-- Check for working C compiler: /cvmfs/oasis.opensciencegrid.org/gm2/prod/external/gcc/v4_9_2/Darwin64bit+14/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Checking whether CXX compiler has -isysroot
-- Checking whether CXX compiler has -isysroot - yes
-- Checking whether CXX compiler supports OSX deployment target flag
-- Checking whether CXX compiler supports OSX deployment target flag - yes
-- Check for working CXX compiler: /cvmfs/oasis.opensciencegrid.org/gm2/prod/external/gcc/v4_9_2/Darwin64bit+14/bin/g++
-- Check for working CXX compiler: /cvmfs/oasis.opensciencegrid.org/gm2/prod/external/gcc/v4_9_2/Darwin64bit+14/bin/g++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- full qual e7:prof reduced to e7
-- Product is gm2artexamples v3_00_00 e7:prof
-- Module path is /cvmfs/oasis.opensciencegrid.org/gm2/prod/external/art/v1_13_01/Modules;/cvmfs/oasis.opensciencegrid.org/gm2/prod/external/cetbu
-- set_install_root: PACKAGE_TOP_DIRECTORY is /Users/lyon/first-try.Wrih/srcs/gm2artexamples
-- cet dot version: 3.00.00
-- Building for Darwin d14 x86_64
-- set_install_root: PACKAGE_TOP_DIRECTORY is /Users/lyon/first-try.Wrih/srcs/gm2artexamples
-- Selected diagnostics option CAUTIOUS
-- cmake build type set to Prof in directory <top> and below
--    DEFINE (-D): ;NDEBUG
-- compiler flags for directory <top> and below
--    C++     FLAGS: -O3 -g -gdwarf-2 -fno-omit-frame-pointer -Werror -pedantic -std=c++1y -Wall -Werror=return-type
--    C       FLAGS: -O3 -g -gdwarf-2 -fno-omit-frame-pointer -Werror -pedantic   -Wall -Werror=return-type
-- Boost version: 1.57.0
-- Found the following Boost libraries:
--   chrono
--   date_time
--   filesystem
--   graph
--   iostreams
--   locale
--   prg_exec_monitor
--   program_options
--   random
--   regex
--   serialization
--   signals
--   system
--   thread
--   timer
--   unit_test_framework
--   wave
--   wserialization
-- CPACK_PACKAGE_VERSION is 3.00.00
-- CPACK_PACKAGE_NAME and CPACK_SYSTEM_NAME are gm2 d14-x86_64-prof
-- Configuring done
CMake Warning (dev):
  Policy CMP0042 is not set: MACOSX_RPATH is enabled by default.  Run "cmake
  --help-policy CMP0042" for policy details.  Use the cmake_policy command to
  set the policy and suppress this warning.

  MACOSX_RPATH is not specified for the following targets:

   gm2artexamples_DataObjects_dict
   gm2artexamples_DataObjects_map
   gm2artexamples_HitAndTrackObjects_dict
   gm2artexamples_HitAndTrackObjects_map
   gm2artexamples_Lesson1_HelloWorld1_module
   gm2artexamples_Lesson1_HelloWorld2_module
   gm2artexamples_Lesson1_MyDatumReader_module
   gm2artexamples_Lesson1_ProduceMyLittleDatum_module
   gm2artexamples_Lesson2_makeHits_module
   gm2artexamples_Lesson2_makeRotatedHits_module
   gm2artexamples_Lesson2_makeSimpleTracksFromNewHits_module
   gm2artexamples_Lesson2_makeSimpleTracksFromOldHits_module
   gm2artexamples_Lesson2_readHits_module
   gm2artexamples_Lesson2_readSimpleTracks_module
   test_MyLittleDatumAnalyzer_module
   test_MyLittleDatumProducer_module

This warning is for project developers.  Use -Wno-dev to suppress it.

-- Generating done
-- Build files have been written to: /Users/lyon/first-try.Wrih/build_d14.x86_64

------------------------------------
INFO: Stage cmake successful.
------------------------------------


------------------------------------
INFO: gm2artexamples version 3.00.00 configured.
------------------------------------


------------------------------------
INFO: Stage build.
------------------------------------

Scanning dependencies of target gm2artexamples_DataObjects
[  3%] Building CXX object gm2artexamples/DataObjects/CMakeFiles/gm2artexamples_DataObjects.dir/MyLittleDatum.cc.o
Linking CXX shared library ../lib/libgm2artexamples_DataObjects.dylib
[  3%] Built target gm2artexamples_DataObjects
[  6%] Generating gm2artexamples_DataObjects_dict.cpp, gm2artexamples_DataObjects_map.cpp
--->> genreflex: INFO: Using gccxml from /cvmfs/oasis.opensciencegrid.org/gm2/prod/external/gccxml/v0_9_20140718a/Darwin64bit+14/bin/gccxml
--->> genreflex: INFO: Parsing file /Users/lyon/first-try.Wrih/srcs/gm2artexamples/DataObjects/classes.h with GCC_XML OK
--->> genreflex: INFO: Generating Reflex Dictionary
class artex::MyLittleDatum
class std::vector<artex::MyLittleDatum>
class art::Wrapper<std::vector<artex::MyLittleDatum> >
Scanning dependencies of target gm2artexamples_DataObjects_dict
[  9%] Building CXX object gm2artexamples/DataObjects/CMakeFiles/gm2artexamples_DataObjects_dict.dir/gm2artexamples_DataObjects_dict.cpp.o
Linking CXX shared library ../lib/libgm2artexamples_DataObjects_dict.dylib
[  9%] Built target gm2artexamples_DataObjects_dict
Scanning dependencies of target gm2artexamples_DataObjects_map
[ 12%] Building CXX object gm2artexamples/DataObjects/CMakeFiles/gm2artexamples_DataObjects_map.dir/gm2artexamples_DataObjects_map.cpp.o
Linking CXX shared library ../lib/libgm2artexamples_DataObjects_map.dylib
[ 15%] Built target gm2artexamples_DataObjects_map
Scanning dependencies of target gm2artexamples_HitAndTrackObjects
[ 18%] Building CXX object gm2artexamples/HitAndTrackObjects/CMakeFiles/gm2artexamples_HitAndTrackObjects.dir/Hit.cpp.o
[ 21%] Building CXX object gm2artexamples/HitAndTrackObjects/CMakeFiles/gm2artexamples_HitAndTrackObjects.dir/HitData.cpp.o
Linking CXX shared library ../lib/libgm2artexamples_HitAndTrackObjects.dylib
[ 21%] Built target gm2artexamples_HitAndTrackObjects
[ 25%] Generating gm2artexamples_HitAndTrackObjects_dict.cpp, gm2artexamples_HitAndTrackObjects_map.cpp
--->> genreflex: INFO: Using gccxml from /cvmfs/oasis.opensciencegrid.org/gm2/prod/external/gccxml/v0_9_20140718a/Darwin64bit+14/bin/gccxml
--->> genreflex: INFO: Parsing file /Users/lyon/first-try.Wrih/srcs/gm2artexamples/HitAndTrackObjects/classes.h with GCC_XML OK
--->> genreflex: INFO: Generating Reflex Dictionary
class artex::SimpleTrackData
class std::vector<artex::SimpleTrackData>
class artex::HitData
class std::vector<artex::HitData>
class art::PtrVector<artex::HitData>
class art::Wrapper<std::vector<artex::SimpleTrackData> >
class art::Wrapper<std::vector<artex::HitData> >
class art::Ptr<artex::HitData>
Scanning dependencies of target gm2artexamples_HitAndTrackObjects_dict
[ 28%] Building CXX object gm2artexamples/HitAndTrackObjects/CMakeFiles/gm2artexamples_HitAndTrackObjects_dict.dir/gm2artexamples_HitAndTrackObjec
Linking CXX shared library ../lib/libgm2artexamples_HitAndTrackObjects_dict.dylib
[ 28%] Built target gm2artexamples_HitAndTrackObjects_dict
Scanning dependencies of target gm2artexamples_HitAndTrackObjects_map
[ 31%] Building CXX object gm2artexamples/HitAndTrackObjects/CMakeFiles/gm2artexamples_HitAndTrackObjects_map.dir/gm2artexamples_HitAndTrackObject
Linking CXX shared library ../lib/libgm2artexamples_HitAndTrackObjects_map.dylib
[ 34%] Built target gm2artexamples_HitAndTrackObjects_map
Scanning dependencies of target gm2artexamples_Lesson1_HelloWorld1_module
[ 37%] Building CXX object gm2artexamples/Lesson1/CMakeFiles/gm2artexamples_Lesson1_HelloWorld1_module.dir/HelloWorld1_module.cc.o
```

```
Linking CXX shared library ../lib/libgm2artexamples_Lesson1_HelloWorld1_module.dylib
[ 37%] Built target gm2artexamples_Lesson1_HelloWorld1_module
Scanning dependencies of target gm2artexamples_Lesson1_HelloWorld2_module
[ 40%] Building CXX object gm2artexamples/Lesson1/CMakeFiles/gm2artexamples_Lesson1_HelloWorld2_module.dir/HelloWorld2_module.cc.o
Linking CXX shared library ../lib/libgm2artexamples_Lesson1_HelloWorld2_module.dylib
[ 40%] Built target gm2artexamples_Lesson1_HelloWorld2_module
Scanning dependencies of target gm2artexamples_Lesson1_MyDatumReader_module
[ 43%] Building CXX object gm2artexamples/Lesson1/CMakeFiles/gm2artexamples_Lesson1_MyDatumReader_module.dir/MyDatumReader_module.cc.o
Linking CXX shared library ../lib/libgm2artexamples_Lesson1_MyDatumReader_module.dylib
[ 43%] Built target gm2artexamples_Lesson1_MyDatumReader_module
Scanning dependencies of target gm2artexamples_Lesson1_ProduceMyLittleDatum_module
[ 46%] Building CXX object gm2artexamples/Lesson1/CMakeFiles/gm2artexamples_Lesson1_ProduceMyLittleDatum_module.dir/ProduceMyLittleDatum_module.cc
Linking CXX shared library ../lib/libgm2artexamples_Lesson1_ProduceMyLittleDatum_module.dylib
[ 46%] Built target gm2artexamples_Lesson1_ProduceMyLittleDatum_module
Scanning dependencies of target gm2artexamples_Lesson2_makeHits_module
[ 50%] Building CXX object gm2artexamples/Lesson2/CMakeFiles/gm2artexamples_Lesson2_makeHits_module.dir/makeHits_module.cc.o
Linking CXX shared library ../lib/libgm2artexamples_Lesson2_makeHits_module.dylib
[ 50%] Built target gm2artexamples_Lesson2_makeHits_module
Scanning dependencies of target gm2artexamples_Lesson2_makeRotatedHits_module
[ 53%] Building CXX object gm2artexamples/Lesson2/CMakeFiles/gm2artexamples_Lesson2_makeRotatedHits_module.dir/makeRotatedHits_module.cc.o
Linking CXX shared library ../lib/libgm2artexamples_Lesson2_makeRotatedHits_module.dylib
[ 53%] Built target gm2artexamples_Lesson2_makeRotatedHits_module
Scanning dependencies of target gm2artexamples_Lesson2_makeSimpleTracksFromNewHits_module
[ 56%] Building CXX object gm2artexamples/Lesson2/CMakeFiles/gm2artexamples_Lesson2_makeSimpleTracksFromNewHits_module.dir/makeSimpleTracksFromNew
Linking CXX shared library ../lib/libgm2artexamples_Lesson2_makeSimpleTracksFromNewHits_module.dylib
[ 56%] Built target gm2artexamples_Lesson2_makeSimpleTracksFromNewHits_module
Scanning dependencies of target gm2artexamples_Lesson2_makeSimpleTracksFromOldHits_module
[ 59%] Building CXX object gm2artexamples/Lesson2/CMakeFiles/gm2artexamples_Lesson2_makeSimpleTracksFromOldHits_module.dir/makeSimpleTracksFromOld
Linking CXX shared library ../lib/libgm2artexamples_Lesson2_makeSimpleTracksFromOldHits_module.dylib
[ 59%] Built target gm2artexamples_Lesson2_makeSimpleTracksFromOldHits_module
Scanning dependencies of target gm2artexamples_Lesson2_readHits_module
[ 62%] Building CXX object gm2artexamples/Lesson2/CMakeFiles/gm2artexamples_Lesson2_readHits_module.dir/readHits_module.cc.o
Linking CXX shared library ../lib/libgm2artexamples_Lesson2_readHits_module.dylib
[ 62%] Built target gm2artexamples_Lesson2_readHits_module
Scanning dependencies of target gm2artexamples_Lesson2_readSimpleTracks_module
[ 65%] Building CXX object gm2artexamples/Lesson2/CMakeFiles/gm2artexamples_Lesson2_readSimpleTracks_module.dir/readSimpleTracks_module.cc.o
Linking CXX shared library ../lib/libgm2artexamples_Lesson2_readSimpleTracks_module.dylib
[ 65%] Built target gm2artexamples_Lesson2_readSimpleTracks_module
Scanning dependencies of target +Users+lyon+first-try.Wrih+build_d14.x86_64+gm2artexamples+bin+myLittleDatum_wr.sh
[ 68%] Copying /Users/lyon/first-try.Wrih/srcs/gm2artexamples/test/myLittleDatum_wr.sh to /Users/lyon/first-try.Wrih/build_d14.x86_64/gm2artexampl
[ 68%] Built target +Users+lyon+first-try.Wrih+build_d14.x86_64+gm2artexamples+bin+myLittleDatum_wr.sh
Scanning dependencies of target +Users+lyon+first-try.Wrih+build_d14.x86_64+gm2artexamples+bin+very_simple_test.sh
[ 71%] Copying /Users/lyon/first-try.Wrih/srcs/gm2artexamples/test/very_simple_test.sh to /Users/lyon/first-try.Wrih/build_d14.x86_64/gm2artexampl
[ 71%] Built target +Users+lyon+first-try.Wrih+build_d14.x86_64+gm2artexamples+bin+very_simple_test.sh
Scanning dependencies of target +Users+lyon+first-try.Wrih+build_d14.x86_64+gm2artexamples+test+MyLittleDatum_test.d+MyLittleDatum_test.fcl
[ 75%] Copying fcl/MyLittleDatum_test.fcl to /Users/lyon/first-try.Wrih/build_d14.x86_64/gm2artexamples/test/MyLittleDatum_test.d/MyLittleDatum_te
[ 75%] Built target +Users+lyon+first-try.Wrih+build_d14.x86_64+gm2artexamples+test+MyLittleDatum_test.d+MyLittleDatum_test.fcl
Scanning dependencies of target +Users+lyon+first-try.Wrih+build_d14.x86_64+gm2artexamples+test+MyLittleDatum_test.d+messageDefaults.fcl
[ 78%] Copying fcl/messageDefaults.fcl to /Users/lyon/first-try.Wrih/build_d14.x86_64/gm2artexamples/test/MyLittleDatum_test.d/messageDefaults.fcl
[ 78%] Built target +Users+lyon+first-try.Wrih+build_d14.x86_64+gm2artexamples+test+MyLittleDatum_test.d+messageDefaults.fcl
Scanning dependencies of target +Users+lyon+first-try.Wrih+build_d14.x86_64+gm2artexamples+test+myLittleDatum_wr.sh.d+MyLittleDatum_r.fcl
[ 81%] Copying fcl/MyLittleDatum_r.fcl to /Users/lyon/first-try.Wrih/build_d14.x86_64/gm2artexamples/test/myLittleDatum_wr.sh.d/MyLittleDatum_r.fc
[ 81%] Built target +Users+lyon+first-try.Wrih+build_d14.x86_64+gm2artexamples+test+myLittleDatum_wr.sh.d+MyLittleDatum_r.fcl
Scanning dependencies of target +Users+lyon+first-try.Wrih+build_d14.x86_64+gm2artexamples+test+myLittleDatum_wr.sh.d+MyLittleDatum_w.fcl
[ 84%] Copying fcl/MyLittleDatum_w.fcl to /Users/lyon/first-try.Wrih/build_d14.x86_64/gm2artexamples/test/myLittleDatum_wr.sh.d/MyLittleDatum_w.fc
[ 84%] Built target +Users+lyon+first-try.Wrih+build_d14.x86_64+gm2artexamples+test+myLittleDatum_wr.sh.d+MyLittleDatum_w.fcl
Scanning dependencies of target +Users+lyon+first-try.Wrih+build_d14.x86_64+gm2artexamples+test+myLittleDatum_wr.sh.d+messageDefaults.fcl
[ 87%] Copying fcl/messageDefaults.fcl to /Users/lyon/first-try.Wrih/build_d14.x86_64/gm2artexamples/test/myLittleDatum_wr.sh.d/messageDefaults.fc
[ 87%] Built target +Users+lyon+first-try.Wrih+build_d14.x86_64+gm2artexamples+test+myLittleDatum_wr.sh.d+messageDefaults.fcl
Scanning dependencies of target simple_test
[ 90%] Building CXX object gm2artexamples/test/CMakeFiles/simple_test.dir/simple_test.cc.o
Linking CXX executable ../bin/simple_test
[ 90%] Built target simple_test
Scanning dependencies of target test_MyLittleDatumAnalyzer_module
[ 93%] Building CXX object gm2artexamples/test/CMakeFiles/test_MyLittleDatumAnalyzer_module.dir/MyLittleDatumAnalyzer_module.cc.o
Linking CXX shared library ../lib/libtest_MyLittleDatumAnalyzer_module.dylib
[ 93%] Built target test_MyLittleDatumAnalyzer_module
Scanning dependencies of target test_MyLittleDatumProducer_module
[ 96%] Building CXX object gm2artexamples/test/CMakeFiles/test_MyLittleDatumProducer_module.dir/MyLittleDatumProducer_module.cc.o
Linking CXX shared library ../lib/libtest_MyLittleDatumProducer_module.dylib
[ 96%] Built target test_MyLittleDatumProducer_module
Scanning dependencies of target test_with_boost
[100%] Building CXX object gm2artexamples/test/CMakeFiles/test_with_boost.dir/test_with_boost.cc.o
Linking CXX executable ../bin/test_with_boost
[100%] Built target test_with_boost
      112.06 real        69.95 user        14.34 sys
```

```
------------------------------------
INFO: Stage build successful.
------------------------------------
```

The long output is not shown. Hopefully there will be no compilation errors. If you get some, ask for help.

3.5 Testing

`gm2artexamples` is currently the only product that has unit tests. To try them, just do `ctest` in the `$MRB_BUILDDIR` directory.

```
$ pushd $MRB_BUILDDIR
$ ctest
$ popd
```

```
~/first-try.Wrih/build_d14.x86_64 ~/first-try.Wrih/srcs
Test project /Users/lyon/first-try.Wrih/build_d14.x86_64
    Start 1: very_simple_test.sh
1/5 Test #1: very_simple_test.sh ..............   Passed    0.08 sec
    Start 2: simple_test
2/5 Test #2: simple_test ......................   Passed    0.08 sec
    Start 3: test_with_boost
3/5 Test #3: test_with_boost ..................   Passed    0.10 sec
    Start 4: MyLittleDatum_test
4/5 Test #4: MyLittleDatum_test ...............   Passed    2.52 sec
    Start 5: myLittleDatum_wr.sh
5/5 Test #5: myLittleDatum_wr.sh ..............   Passed    2.00 sec

100% tests passed, 0 tests failed out of 5

Total Test time (real) =   4.78 sec
~/first-try.Wrih/srcs
```

3.6 Running

There are several fcl files you can run for `gm2artexamples`.

```
$ ls $MRB_SOURCE/gm2artexamples/fcl
```

```
CMakeLists.txt
hello1.fcl
hello2.fcl
makeAndReadDatum.fcl
makeAndReadTracksFromOldHits.fcl
makeDatum.fcl
makeHits.fcl
makeHitsRotated.fcl
makeTracksFromNewHits.fcl
makeTracksFromOldHits.fcl
messageservice.fcl
minimalMessageService.fcl
readDatum.fcl
readHits.fcl
readSimpleTracks.fcl
```

Our `art` executable is called `gm2`. FCL files are found by the `$FHICL_FILE_PATH` search path.

```
$ gm2 -c hello1.fcl
```

```
%MSG-i MF_INIT_OK:  05-May-2015 22:46:22 CDT JobSetup
Messagelogger initialization complete.
%MSG
%MSG-w CONFIG:  05-May-2015 22:46:22 CDT JobSetup
Use of services.user parameter set is deprecated.
Define all services in services parameter set.
%MSG
Begin processing the 1st record. run: 1 subRun: 0 event: 1 at 05-May-2015 22:46:22 CDT
Hello, world. From analyze. run: 1 subRun: 0 event: 1
Begin processing the 2nd record. run: 1 subRun: 0 event: 2 at 05-May-2015 22:46:22 CDT
Hello, world. From analyze. run: 1 subRun: 0 event: 2

TrigReport ---------- Event  Summary ------------
TrigReport Events total = 2 passed = 2 failed = 0

TrigReport ------ Modules in End-Path: end_path ------------
TrigReport  Trig Bit#    Visited     Passed     Failed      Error Name
TrigReport     0    0          2          2          0          0 hello

TimeReport ---------- Time  Summary ---[sec]----
TimeReport CPU = 0.000062 Real = 0.000092

Art has completed and will exit with status 0.
```

3.7 Logging in again

At some point, you will want to log out of your machine and log back in later to continue your work. To reconstitute your development environment, you need to,

- Select the release area

```
source /grid/fermiapp/gm2/setup # on gm2gpvm
source /cvmfs/oasis.opensciencegrid.org/gm2/prod/g-2/setup # On Mac
```

- Change directory to your development area

```
cd ~/Development/g-2/first-time  # On my Mac
```

- Run the setup script in local products (this will re-select the chosen g-2 release)

```
source localProducts_gm2_v6_01_00_prof/setup
```

- Extend the environment for the products your build depends upon (don't forget the leading dot)

```
. mrb s
```

Now you are set to build (`mrb b`), run (`gm2 -c FCL_FILE`), and develop.

3.8 Summary

Here is a summary of the commands for `gm2 v6_01_00`.

3.8.1 *To checkout, build and run* `gm2artexamples` *to a new development area*

```
# Log into machine (e.g. gm2gpvm.fnal.gov)

# Select release area
source /grid/fermiapp/gm2/setup  # On gm2gpvm
source /cvmfs/oasis.opensciencegrid.org/gm2/prod/g-2/setup # On Mac

# Create development area
mkdir /gm2/app/users/lyon/first-time # For me on gm2gpvm
mkdir ~/Development/g-2/first-time   # For me on my Mac
cd <THAT_DIRECTORY>

# Setup the release
setup gm2 v6_01_00 -q prof

# Initialize Development area
mrb newDev
source localProducts_gm2_v6_01_00_prof/setup

# Checkout code
cd srcs
mrb g gm2artexamples

# Extend environment with build dependencies
. mrb s

# Build it
mrb b

# Test it
mrb t

# Run it
gm2 -c hello1.fcl
```

3.8.2 *Restoring environment when logging in again later*

Here's what you do to restore your environment

```
# Log into machine (e.g. gm2gpvm.fnal.gov)

# Select release area
source /grid/fermiapp/gm2/setup  # On gm2gpvm
source /cvmfs/oasis.opensciencegrid.org/gm2/prod/g-2/setup # On Mac

# cd to development area
cd /gm2/app/users/lyon/first-time # For me on gm2gpvm
cd ~/Development/g-2/first-time   # For me on my Mac

# Restore basic environment
source localProducts_gm2_v6_01_00_prof/setup

# Extend environment with build dependencies
. mrb s

# Now you can work!! For example
mrb b  # Build it if you've made a change since last time
ctest  # Test it
gm2 -c hello1.fcl # Run it
```

4
Developer Workflow

The steps you follow to develop code is described here. [**Incomplete**]

4.1 Building your code

You must compile and link your code in order to run it. These tasks are called "building". After you have your development environment configured[1], you must then configure further by configuring to build. Do,

[1] To configure the environment, source the $g-2$ setup script (see section SEC GOES HERE) and then source the `setup` script in local products

```
. mrb s
```

The command above parses all of the `product_deps` files for your packages and determines dependencies. It then sets then all up from ups (e.g. `geant4`). You must run this command every time you start or resume a development session (e.g. log out and back in again later). You should not see any errors.

4.1.1 Building for the first time

When you have no build products at all in your build directory (`$MRB_BUILDDIR`), start the build with

```
mrb b
```

That will run `cmake`[2] and then `make`[3] to build your code. The build will stop if there are any `cmake`, compilation, or linker errors.

[2] See http://www.cmake.org/ for more information.
[3] `make` is the standard build tool that determines dependencies, build order, and issues the commands. `make` uses *Makefiles* for configuration and construction. These Makefiles are extremely difficult to write correctly. `cmake` is a tool with a simpler configuration language that will write all of the Makefile's for us.

4.1.2 Running tests

If packages you are building contain tests, you can run them with,

```
pushd $MRB_BUILDDIR    # cd to that directory and push to stack
ctest -j N             # N is number of CPU cores
popd                   # Change back to old directory
```

Where `N` is equal to or less than the number of cores on your machine (use 1 for gm2gpvm; a newer Mac laptop may have as many as 8 cores). The `-j` option is optional, but if you give it the tester can run tests in parallel and will be much faster. Your current directory must be `$MRB_BUILDDIR`.

If a test fails, look in `$MRB_BUILDDIR/Testing/Temporary` for log files.

4.1.3 Re-building incrementally

When you make a change to your code, you need to build it again (an incremental build). The build system can figure out what has changed and only rebuild the modified code and anything that depends on it. You can do this by re-running `mrb b`. Note that this will re-run `cmake` perhaps unnecessarily. See below for faster ways to rebuild.

4.2 Incremental rebuilds

If you have not changed any `CMakeLists.txt` files and you have not added any new header files, you can skip the `cmake` step on an incremental re-build by doing,

```
pushd $MRB_BUILDDIR   # cd to that directory and push to stack
make -j N             # N is number of CPU cores
popd                  # Change back to old directory
```

This may still take a minute or two as `make` has to check each directory for changes (see below for faster methods). Use `-j` to specify the number of cores on your machine to do builds in parallel. On `gm2gpvm`, leave off the `-j` since there is only one core.

4.2.1 Incremental rebuild (super-fast but potentially dangerous)

When `make` runs, it tells you the *target* that is building. If you know the name of the target for your build, you can tell `make` to only make that target. For example,

```
pushd $MRB_BUILDDIR   # cd to that directory and push to stack
make gm2artexamples_Lesson2_makeRotatedHits_module
popd                  # Change back to old directory
```

This technique is somewhat dangerous, as `make` will not build other targets that depend on the one you have changed, possibly leading to an inconsistent and incorrect build. But, if you are changing an art module which cannot have downstream dependencies, then you are safe in only building that module's target.

This partial build is very fast, as you are telling `make` to only build a very small part of the codebase.

4.2.2 Building with `ninja` (amazingly fast and apparently safe)

`ninja` [4] is a build system that replaces `make`. Fortunately, just as `cmake` knows how to create the files necessary for `make`, `cmake` also knows how to create the files for building with `ninja`. `ninja` works on all platforms.

[4] See http://martine.github.io/ninja/

The advantage of `ninja` over `make` is that if you do an incremental build, `ninja` can determine what files need compiling in practically zero time.

For example, if you do a full build, and then do an incremental build (with `mrb b`) changing nothing, the build will take quite awhile to figure out that there is nothing to do. That is because `make` needs to check each directory for updated files. Somehow, `ninja` figures this out a different way.[5]

[5] I think `ninja` polls the file system event logger to determine what files were updated and it will return instantaneously.

Compiling and linking takes time, of course, but you will get there much faster.

Though I have done builds with `make` and `ninja` and see no problems with using `ninja`, currently `ninja` is experimental and you will have to follow some extra steps to use it.

First build with ninja

`ninja` replaces `make`. You can decide to use `ninja` for your build directory. If you've already done a build with `make`, you must zap it (delete it with `mrb z; .  mrb s`) and then redo the full build with `ninja`. Once you've built with `ninja`, you must zap again to go back to `make` for that build directory. The upshot is that you cannot freely switch between `make` and `ninja` for a build directory.

Because `ninja` is experimental, you must set it up explicitly. Before running the build, do (you will need to do this each time you log in),

```
setup ninja v1_5_3a
```

Now, you need to do a full build with,

```
. mrb s
mrb b --generator ninja
```

You won't see much speed up here, as this is a full build. The speed up occurs for incremental builds.

Incremental builds with ninja

You must have done a full build with `ninja` as described in the previous section. If you have logged out and logged back in in the meantime, re-run the `setup ninja` command above.

Now when you change some code and want to do an incremental build, do

```
pushd $MRB_BUILDDIR   # cd to that directory and push to stack
ninja                 # The magic happens
popd                  # Change back to old directory
```

`ninja` will figure out the number of cores you have. `ninja` will determine all of the files that need to be re-compiled and linked with almost no overhead.

5
Using a Mac for Development

Information and instructions for developing and running $g-2$ code on the Mac has moved to its own document. See GM2-doc-2459.

6
Getting Started with the Simulation

This section gives information and instructions on how to get started with the Muon g-2 simulation. If you are brand new to the simulation, then you have several things you need to learn,

- Geant4
- ArtG4
- Gm2RingSim and its associated packages

Let's go through these things one at a time.

6.1 Geant4

`Geant4` is a toolkit for the simulation of particles passing through matter and fields. You can create all manner of apparatuses, shoot particles at it, and see what the particles will do. `Geant4` has extensive physics models that can handle a wide variety of situations. We use `Geant4` to build our Muon g-2 ring with its detectors and then shoot muons into it. `Geant4` figures out how those muons will behave. It can do decays, spin tracking, interactions with the calorimeter crystals and optical photons, etc. It is quite an extensive package. The home page for `Geant4` is at http://geant4.cern.ch/ . There are three main parts of `Geant4`,

Building the apparatus and detectors You must define the shapes and materials that the particles will be passing through. `Geant4` has an extensive library of materials and shapes to choose from. You must also create "sensitive detectors", which are parts of the apparatus where geant will record interactions and energy loss as hits.

Defining "actions" Geant processes the simulation in many steps, including starting and ending events, tracking new particles, and stepping through parts of the simulation. You can add your code in these processes via `actions`.

Examining hits The ultimate goal of the simulation is to record interactions of particles with the sensitive detectors in the apparatus. Such information is the "truth". You must then have code outside of geant that determines the response of the detectors to these hits (usually called the digitization step).

On the Geant home page are various user guides. The best one to look at for a newcomer is the Users's Guide for Application Developers. Part of learning geant is going through the extensive set of examples. Fortunately, we have all of the examples distributed in our g-2 release.

6.1.1 Building and running the Geant4 examples

See section 3.1 and follow those instructions to set up your environment. Note that `Geant4` has its own build system for the examples, so we will not be using the usual g-2 development area. First, we need to chose a release area.

```
$ if [ -r /grid/fermiapp/gm2/setup ]; then  # Does /grid/fermiapp/gm2/setup exist?
$   source  /grid/fermiapp/gm2/setup   # We're on gm2gpvm
$ else
$   source /cvmfs/oasis.opensciencegrid.org/gm2/prod/g-2/setup # We're on a Mac
$ fi
```

```
g-2 software

--> To list gm2 releases, type
ups list -aK+ gm2

--> To use the latest release, do
setup gm2 v6_00_00 -q prof

For more information, see https://cdcvs.fnal.gov/redmine/projects/g-2/wiki/ReleaseInformation
```

We will only set up `geant4` along with the `cmake` build system, which is all we need to run the examples (below are the latest versions of geant and cmake we have in the release),

```
$ setup geant4 v4_9_6_p04a -q prof
$ setup cmake v3_2_1
```

```
ERROR: Version conflict -- dependency tree requires versions conflicting with current setup of product
```

Let's make a directory to do some work in.

```
$ if [ -r /gm2/app/users/$USER ]; then  # Does /gm2/app/users/YOU exist?
$   # It does, let's use /gm2/app/users/$USER/first-try followed by random letters for uniqueness
$   TMPDIR=`mktemp -d /gm2/app/users/$USER/geant-ex.XXXX`
$ else
$     # We're not on gm2gpvm, let's just make a directory in your home area (hope there's room!)
$     TMPDIR=`mktemp -d ~/geant-ex.XXXX`
$ fi
$
$ # Change directory there
$ cd $TMPDIR
```

You can find the geant examples at, `$GEANT4_DIR/source/geant4.9.6.p04/examples`,

```
$ ls $GEANT4_DIR/source/geant4.9.6.p04/examples
```

```
CMakeLists.txt
GNUmakefile
History
README
README.HowToRun
advanced
basic
extended
novice
```

The `README` file describes the different examples. The build instructions here are based on the `README.HowToRun` file. See that file for more information.

Let's try to build and run the `N05` example in the `novice` directory.

First, we need to make a build area,

```
$ mkdir n05-build
$ cd n05-build
```

Now we run `cmake`[1] with some parameters to set up the build system.

[1] On the Mac, you may see a message about using the AppleClang C compiler. That is not a problem because Geant is all C++ and so the C compiler will not be used.

```
$ export CMAKE_PREFIX_PATH=$GEANT4_FQ_DIR
$
$ cmake -DCMAKE_BUILD_TYPE=Debug \
$       -DCMAKE_CXX_COMPILER=$GCC_FQ_DIR/bin/g++ \
$       -DCMAKE_CXX_FLAGS="-std=c++1y" \
$       $GEANT4_DIR/source/geant4.9.6.p04/examples/novice/N05
```

```
-- The C compiler identification is AppleClang 6.1.0.6020049
-- The CXX compiler identification is GNU 4.9.2
-- Check for working C compiler: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/cc
-- Check for working C compiler: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Checking whether CXX compiler has -isysroot
-- Checking whether CXX compiler has -isysroot - yes
-- Checking whether CXX compiler supports OSX deployment target flag
-- Checking whether CXX compiler supports OSX deployment target flag - yes
-- Check for working CXX compiler: /cvmfs/oasis.opensciencegrid.org/gm2/prod/external/gcc/v4_9_2/Darwin64bit+14/bin/g++
-- Check for working CXX compiler: /cvmfs/oasis.opensciencegrid.org/gm2/prod/external/gcc/v4_9_2/Darwin64bit+14/bin/g++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/lyon/geant-ex.F05T/n05-build
```

And now we run `make`,

```
$ make
```

```
Scanning dependencies of target exampleN05
[  6%] Building CXX object CMakeFiles/exampleN05.dir/exampleN05.cc.o
[ 12%] Building CXX object CMakeFiles/exampleN05.dir/src/ExN05CalorimeterHit.cc.o
[ 18%] Building CXX object CMakeFiles/exampleN05.dir/src/ExN05CalorimeterSD.cc.o
[ 25%] Building CXX object CMakeFiles/exampleN05.dir/src/ExN05DetectorConstruction.cc.o
[ 31%] Building CXX object CMakeFiles/exampleN05.dir/src/ExN05EMShowerModel.cc.o
[ 37%] Building CXX object CMakeFiles/exampleN05.dir/src/ExN05EnergySpot.cc.o
[ 43%] Building CXX object CMakeFiles/exampleN05.dir/src/ExN05EventAction.cc.o
[ 50%] Building CXX object CMakeFiles/exampleN05.dir/src/ExN05EventActionMessenger.cc.o
[ 56%] Building CXX object CMakeFiles/exampleN05.dir/src/ExN05ParallelWorldForPion.cc.o
[ 62%] Building CXX object CMakeFiles/exampleN05.dir/src/ExN05PhysicsList.cc.o
[ 68%] Building CXX object CMakeFiles/exampleN05.dir/src/ExN05PiModel.cc.o
[ 75%] Building CXX object CMakeFiles/exampleN05.dir/src/ExN05PionShowerModel.cc.o
[ 81%] Building CXX object CMakeFiles/exampleN05.dir/src/ExN05PrimaryGeneratorAction.cc.o
[ 87%] Building CXX object CMakeFiles/exampleN05.dir/src/ExN05RunAction.cc.o
[ 93%] Building CXX object CMakeFiles/exampleN05.dir/src/ExN05SteppingAction.cc.o
[100%] Building CXX object CMakeFiles/exampleN05.dir/src/ExN05SteppingActionMessenger.cc.o
Linking CXX executable exampleN05
[100%] Built target exampleN05
```

There will now be an executable `example05` in the build directory. All of the necessary files you need to run (`.in`, `.mac`, `.gdml`) will also be copied to the build directory. Files with `.in` are input macro files. You can run them with (for this example),

```
$ ./example05 example05.in > out   # There is lots of output, so redirect
$ less out  # Examine the output
```

You can also run in interactive mode. This mode will allow you to see the visualizations. For example,

```
$ ./example05

# You will now be at the Idle> prompt
# First, let's run "vis.mac" to set up visualization
Idle> control/execute vis.mac

# Now run some commands.  Best to have another window so you can
# look at example05.in for hints
Idle> /gun/particle e-
Idle> /gun/energy 1 GeV
Idle> /gun/position 0 0 0
Idle> /gun/direction 0 .6 1.
Idle> /run/beamOn 1

# You will see one particle shot into the apparatus
# You can end with exit
Idle> exit
```

Be sure to look at the code and understand what it is doing.

If you want to try a different example, use its directory on the last line of the call to `camke` above.

If you want to alter the example code, then you will have to copy the source code directory to your own directory. Build it the same way as above, but with the last line of the `cmake` call pointing to your source directory.

7 Running the simulation

This section gives you very brief instructions on how to build and run the `gm2ringsim` simulation. More details will be coming in future versions of this document.

Be sure you are familiar with the basics in section 3.

7.1 Component packages in the simulation

Our simulation code is made up of four packages:

- `artg4` - serves as the interface between the `art` framework and `geant4`.
- `gm2geom` - a prototype geometry server
- `gm2dataproducts` - data products used for emitted by the simulation
- `gm2ringsim` - the simulation code itself

7.2 Using a base release

See section 2.6 for the meaning of point and base releases. The base release (e.g `v5_00_00`) only has libraries and executables for the external programs. Therefore to run the simulation from a base release, you must build all of the component packages yourself.

7.3 Using a point release

See section 2.6 for the meaning of point and base releases. With a point release, you may use some or all of the component packages out of the release instead of building them yourself. You should check the `CHANGELOG` (e.g. `less $GM2RINGSIM_DIR/CHANGELOG`) to make sure that the packages were built with the features you want. If so, then simply run a FCL file with `gm2`; no need to build anything or even set up a development area.

If you need to build a package because you want to run something even newer than what was released or you have changes, then follow the dependency tree. In section 7.1, we see the list of components. This was purposefully written to show dependencies from bottom up. E.g. `gm2ringsim` is at the bottom. If you only need to change `gm2ringsim` then you only need to checkout out and build your version of `gm2ringsim`. `gm2ringsim` depends on `gm2dataproducts`, so if you change something in `gm2dataproducts` you will need to checkout and build `gm2dataproducts` **and** `gm2ringsim`. So the way to read that list of components in section 7.1 is that if you change and build a package, you must also change and build everything below it on the list.

7.4 FCL files for the simulation

There are many `fcl` files that you can use to run the simulation. Here's a list of some of them,

`BeamDiagnosticMuPlus.fcl` Shoot individual muons that go around the ring with a rudimentary particle gun with the fiber harp deployed.

`BeamDiagnosticMuPlusMuonGasGun.fcl` Simulation with fiber harp deployed using the gas gun. The gas gun makes muons randomly appear in the ring right before decay.Since geant does not track muons around the ring, this is a very fast simulation.

`ProductionMuPlus.fcl` Shoot individual muons that go around the ring with the ring in data taking state (e.g. no fiber harp).

`ProductionMuPlusMuonGasGun.fcl` Same as above, but using the muon gas gun. Very fast simulation.

`beamtransport_gun.fcl` Muons are not tracked around the ring. Instead, the position and momentum of the muon is calculated using the beam equations of motion and the muon appears in the ring just before it decays. A very accurate and fast simulation.

`inflector_gun.fcl` A very slow but accurate simulation of muons going through the inflector and around the ring.

8 Writing Source Code

Warning: This section needs to be reviewed and cleaned up.

Your source code lives within a git project checked out to your development area's `srcs` directory. The project has a top level directory[1] that contains the "top level" `CMakeLists.txt` file along with various subdirectories. Code with a common purpose should live in a particular subdirectory.[2] You may mix headers (`.h`, `.hh`), implementation (`.cc`, `.cpp`), and configuration (`.fcl`) files all in the same subdirectory.

[1] For example, the `gm2ringsim` project would get checked out to `srcs/gm2ringsim`, which is the "top level" directory.

[2] Examine `gm2ringsim` for more examples.

8.1 Top level *CMakeLists.txt* file

The top level `CMakeLists.txt` file lives in your top level project directory (e.g. `srcs/gm2ringsim/CMakeLists.txt`). It has the main directives that tells CMake how to build your project.

Below is a representative top level `CMakeLists.txt` file.[3] The `mrb newProduct` command will create a skeleton file for you.

[3] There are five main parts of the file (roughly in order in the file)...

- Defining the project
- Loading CMake macros and setting the CMake environment
- Setting compiler options
- Specifying external packages that will be used
- Specifying subdirectories that contain a `CMakeLists.txt` file and, perhaps, code to build

```
# Ensure we are using a moden version of CMake
CMAKE_MINIMUM_REQUIRED (VERSION 2.8)

# Project name - use all lowercase
PROJECT (gm2analyses)

# Define Module search path
set( CETBUILDTOOLS_VERSION $ENV{CETBUILDTOOLS_VERSION} )
if( NOT CETBUILDTOOLS_VERSION )
  message( FATAL_ERROR
         "ERROR:␣setup␣cetbuildtools␣to␣get␣the␣cmake␣modules" )
endif()
set( CMAKE_MODULE_PATH $ENV{CETBUILDTOOLS_DIR}/Modules
                                                ${CMAKE_MODULE_PATH} )

# art contains cmake modules that we use
set( ART_VERSION $ENV{ART_VERSION} )
if( NOT ART_VERSION )
  message( FATAL_ERROR
         "ERROR:␣setup␣art␣to␣get␣the␣cmake␣modules" )
endif()
set( CMAKE_MODULE_PATH $ENV{ART_DIR}/Modules
                                          ${CMAKE_MODULE_PATH} )

# Import the necessary macros
include(CetCMakeEnv)
include(BuildPlugins)
include(ArtMake)
include(FindUpsGeant4)

# Configure the cmake environment
cet_cmake_env()

# Set compiler flags
cet_set_compiler_flags( DIAGS VIGILANT WERROR
   EXTRA_FLAGS -pedantic
   EXTRA_CXX_FLAGS -std=c++11
)

cet_report_compiler_flags()

# Set include and library search paths (the version numbers
# are minimum -  if actual version of product is below specified,
# will get error)

# Everyone should include these
find_ups_product(cetbuildtools v3_07_08)
find_ups_product(art v1_08_10 )
find_ups_product(fhiclcpp v2_17_12)
find_ups_product(messagefacility v1_10_26)

# This project uses code from gm2ringsim,
# gm2dataproducts, and gm2geom
find_ups_product(gm2ringsim v1_00_00)
find_ups_product(gm2dataproducts v1_00_00)
find_ups_product(gm2geom v1_00_00)

# This project uses code from Root
find_ups_root(v5_34_12)

# Make sure we have gcc
cet_check_gcc()

# Macros for art_make and simple plugins (must go after
# find_ups lines)
include(ArtDictionary)

# Specify subdirectories to build
add_subdirectory( ups )  # Every project needs a ups subdirectory
add_subdirectory( DisplayDataProducts )
add_subdirectory( calo )
add_subdirectory( fcl )
add_subdirectory( test )
add_subdirectory( util )

# Packaging facility - required for deployment
include(UseCPack)
```

8.1.1 *When you need to add/change a line in top level `CMakeLists.txt`*

There are two situations for which you will have to alter the top level `CMakeLists.txt` file:

If you add, delete, or rename a subdirectory If you add a subdirectory, you must write a corresponding `add_subdirectory(dirName)` directive.[4] If you delete a directory, you must remove its corresponding `add_subdirectory` line. If you rename a directory, you must edit its corresponding `add_subdirectory` line to reflect the change. If you do not follow these steps, then some code may not build (without an error, so this mistake will be hard to find) or you may get an error when CMake tries to build a directory that no longer exists.

[4] The `add_subdirectory` directory tells CMake to go into that subdirectory and build code there. If you don't have the `add_subdirectory` then CMake won't look in the subdirectory at all.

You use code from an external project If you use code from an external project, you may need to add a corresponding `find_ups_product` or similar line.[5]

[5] See section 8.8 for instructions.

8.2 *Organizing Source Code*

The build system we use is quite flexible and you can organize your code in many ways. You may be used to having all of your header files in an `include` directory with the `.cc` files in other directories. This artificial separation is unnecessary. You may group files together any way you like and may have header files and implementation files in the same directory. Typically, it is best to group files by topic or functionality.

8.3 *Writing Modules*

Modules are plugins to art that perform certain functions (analyzers, producers, filters, and output modules). See section 10 of the Art Work Book[6] for more information. Only reminders will be given here.

[6]

You should use `artmod` to write the skeleton of the module. Do `artmod --help-types` to see the list of module types it will make. Then just run it, giving the name of the class you want including any namespace specification. For example,

```
 artmod producer tracking:TrackFinder
 artmod analyzer gm2analysis::CalorimeterDiags
```

Remember that you specify the class name, not the file name (so do not give `_module` in the name).

8.4 Writing Services

TODO

8.5 Writing Input Source Modules

TODO

8.6 Directory level `CMakeLists.txt` file

If your subdirectory (e.g. `srcs/gm2analyses/strawTracker`) has anything to build, has header files, or has further subdirectories, then it must have a `CMakeLists.txt` file (and a corresponding `add_subdirectory` line in the `CMakeLists.txt` from the directory above - see Sec. 8.1.1).[7] If your subdirectory has code to build, then the directory `CMakeLists.txt` file needs to have

[7] The directory level `CMakeLists.txt` file is different from the top level `CMakeLists.txt` file. The latter is in your project top level directory, like `srcs/gm2analyses`. The former is in a subdirectory of that top level and is described in this section.

```
art_make( )
```

A directory with no `.cc` or `.cpp` files has no code to build and so does not get an `art_make` line in the directory `CMakeLists.txt` file.

See the next section (Sec. 8.6.1) for arguments to the `art_make`. You should call `art_make` only once per `CMakeLists.txt` file.

If your subdirectory has header files, then those have to be copied to the release area when one runs `mrb install`. To do that, you need a line the directory `CMakeLists.txt` file with

```
install_headers( )  # No arguments
```

If your subdirectory has fcl files, then those need to be copied to the build area as well as the release area. There is some scripting involved to do that (put the following in the directory `CMakeLists.txt` file),

```
# install all *.fcl files in this directory to the release area
file(GLOB fcl_files *.fcl)
install( FILES ${fcl_files}
         DESTINATION ${product}/${version}/fcl )

# Also install to the build area
foreach(aFile ${fcl_files})
  get_filename_component( basename ${aFile} NAME )
  configure_file(
         ${aFile} ${CMAKE_BINARY_DIR}/${product}/fcl/${basename}
         COPYONLY )
endforeach(aFile)
```

If your subdirectory has futher subdirectories with code to build, then you need an `add_subdirectory( dirName )` line for each subdirectory.

8.6.1 *Arguments to* `art_make`

You can find documentation for `art_make` in its source code at `$ART_DIR/Modules/ArtMake.cmake`. Basically, you need to specify what libraries to link against when you use external code.[8] If you don't use any external code, then you will have no arguments to `art_make`. It will tell CMake to build all regular source, modules, services, and input sources in the directory. If you do use external code, then you have four choices,

[8] See Sec. 8.8 for how to tell if you are using external code.

- If the source file using external code is a regular source (not a module, not a service, not an import source), then you need

```
    art_make(
          LIB_LIBRARIES
          library1
          library2   # if needed
    )
```

- If the source file using the external code is a module source (e.g. `analyze_my_hits_module.cpp`) then you need

```
    art_make(
          MODULE_LIBRARIES
          library1
          library2   # if needed
      )
```

- If the source file using the external code is a service source (e.g. `analyze_my_hits_service.cpp`) then you need

```
    art_make(
          SERVICE_LIBRARIES
          library1
          library2   # if needed
      )
```

- If the source file using the external code is source code for an input source
  (e.g. `midas_source.cpp`) then you need

```
    art_make(
        SOURCE_LIBRARIES
        library1
        library2   # if needed
     )
```

If you have a mixture of sources in your directory, you can string the calls together. For example,[9]

```
    art_make (
        LIB_LIBRARIES
          ${ROOT_GPAD}
        MODULE_LIBRARIES
          gm2analyses_util
          gm2analyses_strawtracker_util
         )
```

[9] In the example to the left, regular sources get linked against Root's `libGpad.so` (see Sec. 8.8.2) and modules get linked against code built in the `srcs/gm2analyses/util` and `srcs/gm2analyses/strawtracker/util` directories (see Secs. 8.8.4 and 8.8.5).

Note that it does not hurt for code to build against a library that it doesn't need. So if you have five modules and only one needs to link against a library, put that library in the `MODULE_LIBRARIES` section. The one that needs it will link against it and the four that don't won't care.

8.7 Libraries produced from building

Every directory in your project that has code to build generates at least one library.[10] Say, for example, you have a directory called `gm2analyses/calo`. Regular sources (not modules, services, nor input sources) get compiled and the objects go into a library called `libgm2analyses_calo.so` (the name is the directory path with slashes replaced by underscores). Each module in the directory gets its own library. For example, if there is a module in that directory called `Analyze_Calo_module.cc` then that code will go into a library called `libgm2analyses_calo_Analyze_Calo_module.so`. A similar thing happens for services and input sources. Therefore, one directory of code may produce several libraries. The `art_make` directive in the directory `CMakeLists.txt` file tells the build system to build code and make the corresponding libraries.

[10] An important note, if your directory **only** has header files in it (should be a rare situation for code written by users), then no library will be produced (because there is no code to build - the header files are all included by other source code). You still need the directory level `CMakeLists.txt` file for the `install_headers()` directive, but do not do `art_make`. See Sec. 8.6.

8.8 Using External Code (Linking)

Your code is almost never self-contained, especially when writing within the Art framework. You may use functions and classes from external libraries, like Root and Geant4. You may use algorithms, data products, and other functionalities from other projects, like gm2ringsim. You may use objects defined in other directories in your project. If you are writing an art module or service, you may use objects defined in the same directory, but in a different file from the module or service. All of these examples are "external code".

Art uses *dynamic linking*, which means that the art executable (ours is called gm2) has very little code in it. Instead, it loads all of the libraries it needs at runtime. The other style is *static linking* where the executable has embedded in it all of the libraries it needs. Dynamic linking, as the name suggests, allows for flexibility with one executable able to load a variety of different libraries decided upon at runtime with the configuration file. There is, however, overhead in dynamic loading typically experienced as slow start-up time of the program. Static linking produces an executable with all of the libraries built in - so there is little flexibility in terms of functionality. But the start up time is much faster. Static linking typically leads to many copies of executables for the different functionalities, resulting in duplication of libraries that are in common. For maximum flexibility and non-duplication of libraries, art loads everything dynamically.

HOW DO YOU KNOW WHEN YOU ARE USING EXTERNAL CODE? An easy indicator is when you have a #include for a header file. For each #include, you need to think and perhaps add a corresponding link directive in a CMakeLists.txt file.[11] If you forget to link to a library that you need, you will get a missing symbol error when you try to run. This section will explain how to figure out these situations and actions you need to take.

[11] Remember the two types of CMakeLists.txt files: "top level" and "directory level". The former (see Sec. 8.1) is the potentially big file at the top level of your project. The latter (see Sec. 8.6) is the smaller file in the directory with your actual source code files.

8.8.1 Includes for system headers and base art headers

System headers, like #include <string> do not require any special directives for linking. You get them for free.

Headers in art, fhiclcpp, and messagefacility do not require anything in your directory level CMakeLists.txt file. The corresponding libraries are automatically loaded by the art executable. Your top level CMakeLists.txt file must contain the following lines,[12]

```
...
cet_report_compiler_flags()
...
find_ups_product(art v1_08_10 )
find_ups_product(fhiclcpp v2_17_12)
find_ups_product(messagefacility v1_10_26)
...
```

[12] These lines add header file directories to the compiler include search path (e.g. without them, you will get a compilation error that header files cannot be found).

8.8.2 Includes for Root headers

Including a header from Root is a little unusual because you do not have to give a path in the include, e.g. `#include "TCanvas.h"` (not `#include "root/TCanvas.h"`). If you include a header from Root, you will also need to link to the corresponding Root library. First, in the top level `CMakeLists.txt` file, you must have,[13]

```
...
cet_report_compiler_flags()
...
find_ups_root(v5_34_12)
...
```

[13] That `find_ups_root` line adds the Root headers to the compiler include search path and creates CMake variables corresponding to each Root library.

If you look at the code for the `find_ups_root` CMake macro at `$CETBUILDTOOLS/Modules/FindUpsRoot.cmake` you will see lines like,[14]

```
find_library(ROOT_GLEW NAMES GLEW PATHS ${ROOTSYS}/lib
                         NO_DEFAULT_PATH)
find_library(ROOT_GPAD NAMES Gpad PATHS ${ROOTSYS}/lib
                         NO_DEFAULT_PATH)
find_library(ROOT_GRAF NAMES Graf PATHS ${ROOTSYS}/lib
                         NO_DEFAULT_PATH)
find_library(ROOT_GRAF3D NAMES Graf3d PATHS ${ROOTSYS}/lib
                         NO_DEFAULT_PATH)
```

[14] These lines define the CMake variables that correspond to Root libraries. You use them in the directory level `CMakeLists.txt` file to tell CMake to link against that library.

To determine the Root library you need, look up the Root object in the Root documentation at http://root.cern.ch/drupal/content/reference-guide (select the appropriate version of Root - usually the PRO version). Find the class name from the list and click on it. On the new page, on the very right hand side in a little greyed out box it will say the library that corresponds to that Root object. For example, if you `#include "TCanvas.h"` you need to link against the `libGpad` library. The CMake variable name will in general be the name of the library, all upper case, with the `lib` replaced by `ROOT_`. So `libGpad` → `${ROOT_GPAD}`.

In your directory level `CMakeLists.txt` file, you will have the `art_make` directive. Add the appropriate CMake variable corresponding to the Root library you need. See Sec. 8.6.1 for where to put such items in the arguments. For example,[15]

```
        art_make (
           LIB_LIBRARIES
              ${ROOT_GPAD}
           MODULE_LIBRARIES
              ${ROOT_TREE}
              ${ROOT_TVMA}
```

[15] In the example left, regular sources are linked against `libGpad.so` while modules are linked against `libTree.so` and `libTVMA.so`.

```
        )
```

8.8.3 Includes for GEANT headers

To include a header file from Geant4, requires you to have `Geant4/` in the header path, for example `#include "Geant4/G4Track.hh"`. If you include such headers in your code, then you will also need to link against the Geant4 libraries. First, in your top level `CMakeLists.txt` file, you must have,

```
...
cet_report_compiler_flags()
...
find_ups_geant4(v4_9_6_p02)
...
```

That line adds the Geant4 headers to the compiler include search path and creates the CMake variables `${G4_LIB_LIST}` and `${XERCESLIB}`. For any Geant4 header, just add those CMake variables to the `art_make` directive in your directory `CMakeLists.txt` file. See Sec. 8.6.1 for where to put such items in the arguments. For example,
`srcs/gm2ringsim/calo/CMakeLists.txt` has, in part,[16]

```
art_make(
    LIB_LIBRARIES
        gm2geom_calo
        gm2geom_station
        artg4_material
        artg4_util
        ${XERCESCLIB}
        ${G4_LIB_LIST}
    SERVICE_LIBRARIES
        gm2ringsim_calo
      )
```

[16] If you are curious, you can see where `G4_LIB_LIST` is defined in `$CETBUILDTOOLS_DIR/Modules/FindUpsGeant4.cmake`. `XERCESLIB` goes with Geant.

8.8.4 Includes for headers in the project

The `#include` directive should include the path to the header file, including the name of the project even if the header is in the same directory as the source, though you could just give the header file name. For example, if `CaloHitSD.hh` is in the `gm2ringsim/calo` directory, then `CaloHitSD.cc`, when it includes `CaloHitSD.hh`, can do either

```
#include "CaloHitSD.hh"
```

or

```
#include "gm2ringsim/calo/CaloHitSD.hh"
```

The latter is preferred as it is clearer, but if you change the name of the directory, you must change the include as well.

If you have a regular source file and it includes a header that is present in the same directory, then you do not need to do anything to the `CMakeLists.txt` files. If you have a module, service, or input source file and it includes a header that is present in the same directory, then you need to link against the library for that directory. You do not need to add anything to the top level `CMakeLists.txt` file. To the directory `CMakeLists.txt` file, you must add the library. See Sec. 8.6.1 for where to put such items in the arguments. For example, `srcs/gm2ringsim/calo/CMakeLists.txt` has, in part,[17]

```
 art_make(
     LIB_LIBRARIES
         gm2geom_calo
         gm2geom_station
         gm2ringsim_station
         artg4_material
         artg4_util
         ${XERCESCLIB}
         ${G4_LIB_LIST}
     SERVICE_LIBRARIES
         gm2ringsim_calo
       )
```

[17] In the left example, services in that directory are linked against the library that gets created from the regular sources, namely `libgm2ringsim_calo.so`. You can predict the name of the library by taking the source directory (e.g. `gm2ringsim/calo`) and replacing the slashes by underscores.

If any source file uses a header that is present in a different directory in your project, then you must link against that library. In the example above, code in the `gm2ringsim/calo` directory includes code from `gm2ringsim/station`, and hence `gm2ringsim_station` is present in the arguments of `art_make`.

An important exception to these instructions is if the directory with the header file contains **only** header files. In that case, that directory produces no libraries and you do not have to change the directory `CMakeLists.txt` file.

8.8.5 Includes for headers in other projects

If you have a source file (regular, module, service, or input source) that uses code from another project, then you need to do some work. An example here is code in `gm2ringsim` uses code from the `gm2geom` and `artg4` projects. The `#include` needs the path to the header file including project name, directory name and header name. For example, `#include "artg4/util/util.hh"`.

In your top level `CMakeLists.txt` file, you need a `find_ups_product` line for the project specifying the project name and a minimum version number. See Sec.8.1 for an example.

In your directory `CMakeLists.txt` file, you need to list the library corresponding to the code you are using. See Sec. 8.6.1 for where to put such items in the `art_make` arguments. For example, `srcs/gm2ringsim/calo/CMakeLists.txt` has, in part,

```
art_make(
    LIB_LIBRARIES
        gm2geom_calo
        gm2geom_station
        artg4_material
        artg4_util
        ${XERCESCLIB}
        ${G4_LIB_LIST}
    SERVICE_LIBRARIES
        gm2ringsim_calo
      )
```

When the regular sources are built, they will be linked against code in `gm2geom/calo`, `gm2geom/station`, `artg4/material`, and `artg4/util`.

An important exception to these instructions is if the directory with the header file contains **only** header files. In that case, that directory produces no libraries and you do not have to change the directory `CMakeLists.txt` file. You still need to have the top level `CMakeLists.txt` file correct as described above.

9
Things You May Do in Your Code

This chapter contains some reminders of common things you do in Muon $g-2$ code.

9.1 Dealing with parameters

The constructor for your module or service has the parameter set as an argument. You can retrieve information from the parameter set and supply defaults if the parameter does not exist as in the example below.

```
gm2ex::CalorimeterDigitizer::CalorimeterDigitizer(
             fhicl::ParameterSet const & p) :
  category_    (p.get<std::string>("category","digi")),
  TAURAMP_     (p.get<float>("TAURAMP", 1.4 /* ns */)),
  TAUDECAY_    (p.get<float>("TAUDECAY", 36.4 /* ns */)),
  PULSELENGTH_ (p.get<int>("PULSELENGTH", 30 /* samples */)),
// ...
```

9.2 Readling enviornment variables

```
 #include <cstdlib>
//...
std::string value = std::getenv("PATH'');
```

The argument to `std::getenv` is a constant character array, not a `std::string`.

9.3 Throwing an exception

See http://mu2e.fnal.gov/public/hep/computing/exceptions.shtml.

```
#include "cetlib/exception.h"
// ...
if ( something ) {
  throw cet::exception(CATEGORY) << "Message\n"
}
```

9.4 *Finding a file*

`cetlib` has a nice facility for searching for files in a path specification. See `$CETLIB_INC/cetlib/search_path.h`.

It may be convenient to specify the search path in a FHICL parameter with the possibility of providing an environment variable. Here is some code that takes a search path through the parameter, but if the first character is a $, it then gets the path through the specified environment variable.

```
gm2util::MetadataFromFile::MetadataFromFile(
          fhicl::ParameterSet const & p) :
    searchPath_  (p.get<std::string>("searchPath", ".")),
    fileName_    (p.get<std::string>("fileName")),
    keyName_     (p.get<std::string>("keyName"))
{
  // Let's parse the search path
  // If the first character is a dollar sign, then the
  // remaining is an environment variable
  if ( searchPath_.at(0) == "$" ) {
    std::string envVar = searchPath_.substr(1);
    char* envValue = std::getenv(envVar.c_str());
    if ( ! envValue ) {
      searchPath_ = ".";
      throw cet::exception("META_DATA_FROM_FILE") <<
       "Environment variable " << envVar << " is not set";
    }

    searchPath_ = std::string(envValue);
  }
}
```

10
Frequently Asked Questions

Some questions are answered here that didn't seem to fit in other sections.

Where is the art source code? The art source code[1] for a particular `gm2` release is accessible in our release area for you to peruse. Set up the the release (see section 3) and look in `$ART_DIR/source/art`.

[1] Never use the source code directory for an `#include` in your code. Instead, just use `#include "art/whatever.h"` and the build system will find it in `$ART_INC`.

Index